ASSIGNMENT NO.5

Title: ECG Anomaly detection using Autoencoders

Aim: Use Autoencoder to implement anomaly detection. Build the model by using:
Import required libraries
Upload / access the dataset
Encoder converts it into latent representation
Decoder networks convert it back to the original input
Compile the models with Optimizer, Loss, and Evaluation Metrics

Theory:

Anomaly detection is examining specific data points and detecting rare occurrences that seem suspicious because they're different from the established pattern of behaviors. Anomaly detection isn't new, but as data increases manual tracking is impractical.

Within this dataset are data patterns that represent business as usual. An unexpected change within these data patterns, or an event that does not conform to the expected data pattern, is considered an anomaly.

There are three main classes of anomaly detection techniques: unsupervised, semi-supervised, and supervised. Essentially, the correct anomaly detection method depends on the available labels in the dataset.

Use Cases:

•Fraud in banking (credit card transactions, tax return claims, etc.), insurance claims (automobile, health, etc.), telecommunications, and other areas is a significant issue for both private business and governments. Fraud detection demands adaptation, detection, and prevention, all with data in real-time.

- Detecting anomalies in medical images and records enables experts to diagnose and treat patients more effectively. Massive amounts of imbalanced data means reduced ability to detect and interpret patterns without these techniques. This is an area that is ideal for artificial intelligence given the tremendous amount of data processing involved.
- Log anomaly detection enables businesses to determine why systems fail by reconstructing faults from patterns and past experiences.

Autoencoders are a type of deep learning algorithm that are designed to receive an input and transform it into a different representation. They play an important part in image construction. Let's learn about autoencoders in detail. They are used to compress the data and reduce its dimensionality. Autoencoders reconstruct our original input given just a compressed version of it.

An Autoencoder is a type of neural network that can learn to reconstruct images, text, and other data from compressed versions of themselves. An Autoencoder consists of three layers:

1. Encoder: The Encoder layer compresses the input image into a latent space representation. It encodes the input image as a compressed representation in a reduced dimension.
2. Code: The Code layer represents the compressed input fed to the decoder layer.
3. Decoder: The decoder layer decodes the encoded image back to the original dimension. The decoded image is reconstructed from latent space representation, and it is reconstructed from the latent space representation and is a lossy reconstruction of the original image.

Applications of autoencoders:

1. Dimensionality reduction

Undercomplete autoencoders are those that are used for dimensionality reduction. These can be used as a pre-processing step for dimensionality reduction as they can perform fast and accurate dimensionality reductions without losing much information. Furthermore, while dimensionality reduction procedures like PCA can only perform linear dimensionality reductions, undercomplete autoencoders can perform large-scale non-linear dimensionality reductions.

2. Generation of image and time series data

Variational Autoencoders can be used to generate both image and time series data. The parameterized distribution at the bottleneck of the autoencoder can be randomly sampled to generate discrete values for latent attributes, which can then be forwarded to the decoder, leading to generation of image data. VAEs can also be used to model time series data like music.

3. Anomaly detection

Undercomplete autoencoders can also be used for anomaly detection. For example-consider an autoencoder that has been trained on a specific dataset P. For any image sampled for the training

dataset, the autoencoder is bound to give a low reconstruction loss and is supposed to reconstruct the image as is.

There are 3 main approaches to detect anomalies:

1. Determination of outliers without previous data information (anomalies). This is analogous to unsupervised clustering.
2. Supervised machine learning issue with usable labels for both regular and anomalous classes. As anomalies are expected to be very few, this could be considered as binary classification with imbalanced data.
3. Learning only the labels of the normal class makes it a semi-supervised approach.

Following are some of the anomaly detection algorithm

1. K-nearest neighbour: k-NN k-NN is one of the simplest supervised learning algorithms and methods in machine learning. It stores all of the available examples and then classifies the new ones based on similarities in distance metrics. In addition, density-based distance measures are good solutions for identifying unusual conditions and gradual trends.
2. Local Outlier Factor (LOF)

The LOF is a key anomaly detection algorithm based on a concept of a local density. It uses the distance between the k nearest neighbors to estimate the density. LOF compares the local density of an item to the local densities of its neighbors. Thus, one can determine areas of similar density and items that have a significantly lower density than their neighbors.

3. K-means

K-means is a very popular clustering algorithm in the data mining area. It creates k groups from a set of items so that the elements of a group are more similar. Just to recall that cluster algorithms are designed to make groups where the members are more similar. In this term, clusters and groups are synonymous.

4. Support Vector Machine (SVM)

A support vector machine is also one of the most effective anomaly detection algorithms. SVM is a supervised machine learning technique mostly used in classification problems. When it comes to anomaly detection, the SVM algorithm clusters the normal data behavior using a learning area. Then, using the testing example, it identifies the abnormalities that go out of the learned area.

5. Neural Networks Based Anomaly Detection

When it comes to modern anomaly detection algorithms, we should start with neural networks. What makes them very helpful for anomaly detection in time series is this power to find out dependent features in multiple time steps. There are many different types of neural networks and they have both supervised and unsupervised learning algorithms.

Difference between Anomaly detection and Novelty Detection.

Aspect	**Anomaly Detection**	**Novelty Detection**
Objective	Identifying abnormal instances within a dataset.	Identifying new, previously unseen instances.
Data Characteristics	Requires a dataset with both normal and abnormal instances.	Typically, trained on a dataset with only normal instances, then tested for novelties.
Training Data	Uses both normal and abnormal data to train the model.	Trains on normal data only, making it unaware of abnormal data.
Evaluation Metrics	Typically uses precision, recall, F1score, etc.	Evaluated based on the model's ability to detect novelties and false positives.
Algorithm Types	One-class SVM, Isolation Forest, Autoencoders, etc.	k-Nearest Neighbours, Local Outlier Factor, One-class SVM (with novelty detection setup), etc.
Data Labelling	Requires labelled data for both normal and abnormal instances.	Typically, doesn't require labelled abnormal data during training.
Applications	Used for fraud detection, network intrusion detection, quality control, etc.	Useful in scenarios where new types of data or events need to be detected.

An autoencoder consists of three components:

- Encoder: An encoder is a feedforward, fully connected neural network that compresses the input into a latent space representation and encodes the input image as a compressed representation in a reduced dimension. The compressed image is the distorted version of the original image.

- Code/ Bottleneck: This part of the network contains the reduced representation of the input that is fed into the decoder.
- Decoder: Decoder is also a feedforward network like the encoder and has a similar structure to the encoder. This network is responsible for reconstructing the input back to the original dimensions from the code.

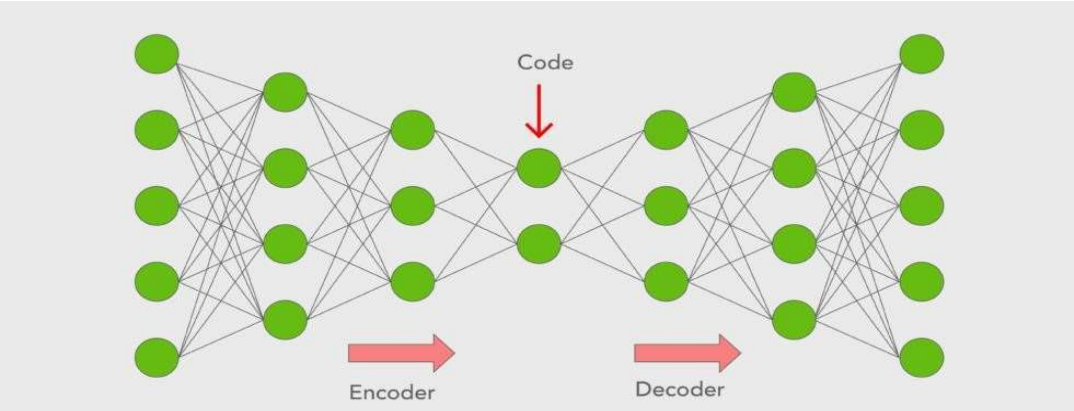


Figure 1: Decoder and Encoder

First, the input goes through the encoder where it is compressed and stored in the layer called Code, then the decoder decompresses the original input from the code. The main objective of the autoencoder is to get an output identical to the input.

Working of Autoencoders:

1. Encoding: The input data is passed through the encoder, which reduces the dimensions and maps it to a lower-dimensional representation.
2. Bottleneck Representation: The encoded representation in the bottleneck layer is a compressed version of the input data, capturing its most important features.
3. Decoding: The decoder takes the bottleneck representation and attempts to reconstruct the original input data.
4. Training: During the training process, the autoencoder tries to minimize the reconstruction error (the difference between the input and the output). This encourages the network to learn a meaningful and compact representation in the bottleneck layer.

Reconstruction is a concept commonly used in the context of autoencoders and other machine learning techniques, especially in the context of unsupervised learning. It refers to the process of generating an output or replica of the input data based on a model's learned internal representation. In the context of autoencoders, which are neural networks designed for feature

learning and data compression, reconstruction specifically refers to generating an output that closely resembles the original input data.

Reconstruction Error refers to the measure of the dissimilarity between the original input data and its reconstructed output. It quantifies how well the autoencoder is able to reproduce the input. A lower reconstruction error indicates that the autoencoder is better at capturing the important features of the data.

There are various ways to calculate the reconstruction error, depending on the type of data and the application:

- For numerical data, commonly used metrics include Mean Squared Error (MSE) or Mean Absolute Error (MAE). These metrics measure the average squared or absolute difference between corresponding elements of the input and reconstructed data.
- For binary data, cross-entropy loss or binary cross-entropy loss is often used to measure the dissimilarity between the original and reconstructed data.
- For image data, structural similarity index (SSIM) and peak signal-to-noise ratio (PSNR) are sometimes used as evaluation metrics.

Minmaxscaler from sklearn.

The **MinMaxScaler** is a data preprocessing technique available in the scikit-learn (sklearn) library, which is a popular machine learning library in Python. MinMaxScaler scales the data to a fixed range, typically between 0 and 1.

The purpose of the MinMaxScaler is to transform the features (columns) of a dataset in such a way that they are all scaled to a specific range, typically between 0 and 1. It is a form of feature scaling, which is important in machine learning to ensure that different features have similar scales and do not unduly influence the learning algorithms.

Formula: The formula for Min-Max scaling is:

X_scaled = (X - X_min) / (X_max - X_min)

- X is the original feature value.
- X_min is the minimum value of the feature in the dataset.
- X_max is the maximum value of the feature in the dataset.
- X_scaled is the scaled value of the feature.

The **train_test_split()** method is used to split our data into train and test sets. First, we need to divide our data into features (X) and labels (y). The dataframe gets divided into X_train, X_test, y_train, and y_test. X_train and y_train sets are used for training and fitting the model. This process is essential in machine learning to evaluate the performance of a model. The training set is used to train the model, and the testing set is used to assess the model's generalization performance on unseen data.

- Train set: The training dataset is a set of data that was utilized to fit the model. The dataset on which the model is trained. This data is seen and learned by the model.
- Test set: The test dataset is a subset of the training dataset that is utilized to give an accurate evaluation of a final model fit.

Here's an explanation of how train_test_split works:

1. Importing the Function:

To use train_test_split, you first need to import it from scikit-learn:

```
from sklearn.model_selection import train_test_split
```

2. Parameters:

The function takes several parameters:

- X: This is the feature matrix or dataset that you want to split.
- y: If you have a target variable (labels), you can provide it here. This is optional.
- test_size: This is the proportion of the dataset to include in the testing set. It can be a floating-point number (e.g., 0.2 for 20% of the data) or an integer (representing the number of samples in the testing set).
- train_size: This parameter is the complement of test_size. If you don't specify test_size, train_size is used instead.
- random_state: A seed value to ensure reproducibility. If you use the same random_state value, you will get the same data split every time. It is optional.

3. Splitting the Data:

The train_test_split function takes your dataset and randomly shuffles it, then divides it into a training set and a testing set according to the specified proportions. The training set typically contains the majority of the data, while the testing set contains a smaller portion.

4. Output: The function returns four datasets (or two if you're not using labels y):

- X_train: The feature matrix for the training set.
- X_test: The feature matrix for the testing set.
- y_train: The target values for the training set (if y was provided).
- y_test: The target values for the testing set (if y was provided).

Every anomaly has an **anomaly score** assigned to it. That score indicates how anomalous the data point is, which makes it possible to define its severity compared to other anomalies. This page gives you a high-level explanation of the critical factors considered for calculating anomaly scores, how the scores are calculated, and how renormalization works.

Three factors can affect the initial anomaly score of a record:

- single bucket impact,
- multi-bucket impact,
- anomaly characteristics impact.

The process of assigning anomaly scores in anomaly detection typically involves the following steps:

1. Model Training
2. Scoring
3. Thresholding
4. Anomaly Identification

TensorFlow Datasets is a collection of datasets ready to use, with TensorFlow or other Python ML frameworks, such as Jax. All datasets are exposed as **tf.data.Datasets** , enabling easy-to-use and high-performance input pipelines. To get started see the guide and our list of datasets.

In TensorFlow, a tf.data.Dataset is an API for building efficient and scalable data input pipelines for machine learning and deep learning workflows. It is designed to handle large datasets, perform data preprocessing, and feed data to machine learning models in a highly optimized and parallelized manner. tf.data.Dataset is a crucial component for efficient data handling in TensorFlow, and it offers several features and capabilities

The **Keras optimizer** ensures that appropriate weights and loss functions are used to keep the difference between the predicted and actual value of the neural network learning model optimized. There are various types of Keras optimizers available to choose from.

Keras optimizer helps us achieve the ideal weights and get a loss function that is completely optimized. One of the most popular of all optimizers is gradient descent. Various other keras optimizers are available and used widely for different practical purposes. There is a provision of various APIs provided by Keras for implementing various optimizers of Keras.

Types of Keras Optimizers

- Adagrad: This optimizer of Keras uses specific parameters in the learning rates. It has got its base of the frequencies made in the updates by the value of parameters, and accordingly, the working happens.

- Adam: This optimizer stands for Adaptive Moment estimation. This makes the adam algorithm; the gradient descent method is upgraded for the optimization tasks. It requires less memory and is very efficient.
- Nadam: This optimizer makes use of the Nadam algorithm. I stand for Nesterov and adam optimizer, and the component of Nesterov is more efficient than the previous implementations.
- Adamax: It is the adaption of the algorithm of Adam optimizer hence the name Adam max. The base of this algorithm is the infinity norm.
- RMSprop: It stands for Root mean Square propagation. The main motive of the RMSprop is to make sure that there is a constant movement in the average calculation of the square of gradients, and the performance of the task of division for gradient upon the root of average also takes place.

Keras Dense Layer and Dropout Layer

1. **Dense Layer:**

A Dense layer is a standard fully connected layer in a neural network. It connects every neuron in the current layer to every neuron in the subsequent layer. The term "dense" refers to the fact that all neurons are connected. Each connection has a weight associated with it, and the layer performs a linear transformation followed by an activation function on its input.

Following is an example of how to create a Dense layer in Keras:

```
from tensorflow import keras model
= keras.Sequential()
model.add(keras.layers.Dense(units=128, activation='relu', input_shape=(input_dim,)))
```

2. Dropout Layer:

A Dropout layer is a regularization technique used to prevent overfitting in neural networks. It randomly sets a fraction of the input units (neurons) to zero during each forward and backward pass. This dropout effectively "drops out" some neurons from the network, making it more robust and less prone to overfitting.

In Keras, a "**loss function**" is a critical component in training a neural network. It quantifies the difference between the predicted values generated by the network and the actual target values in your training data. The choice of an appropriate loss function depends on the type of problem you are solving (e.g., regression, classification, etc.).

Mean Squared Logarithmic Error (MeanSquaredLogarithmicError) is a specific loss function used in regression tasks. It is a variation of the Mean Squared Error (MSE) loss function that is particularly useful when the target values cover a wide range and vary

significantly. Mean Squared Error (MSE) calculates the average of the squared differences between predicted and actual values.

The formula for MSLE is as follows:

MSLE = (1/n) * Σ(log(y_pred_i + 1) - log(y_true_i + 1))^2

• `n` is the number of data points.

• `y_pred_i` is the predicted value for data point `i`.

• `y_true_i` is the true (target) value for data point `i`.

A **rectified linear unit (ReLU)** is an activation function that introduces the property of nonlinearity to a deep learning model and solves the vanishing gradients issue. It interprets the positive part of its argument. The diagram below with the blue line is the representation of the Rectified Linear Unit (ReLU), whereas the green line is a variant of ReLU called Softplus. The other variants of ReLU include leaky ReLU, exponential linear unit (ELU) and Sigmoid linear unit (SiLU), etc., which are used to improve performances in some tasks.

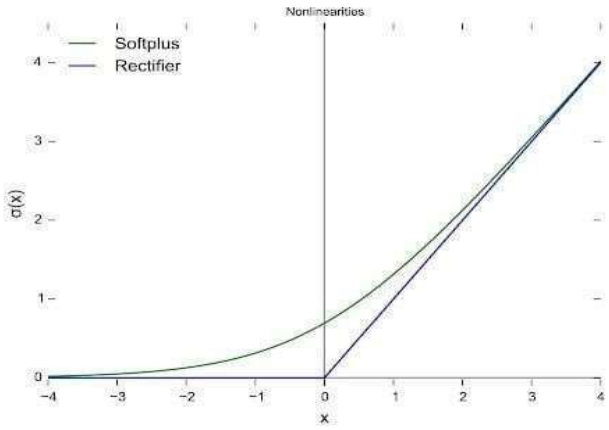


Figure 2: ReLU

The ReLU activation function is differentiable at all points except at zero. For values greater than zero, we just consider the max of the function. This can be written as: f(x)=max(0,x)

Steps/ Algorithm

Dataset link and libraries:

Dataset: http://storage.googleapis.com/download.tensorflow.org/data/ecg.csv Libraries required:

Pandas and Numpy for data manipulation Tensorflow/Keras for Neural Networks

Scikit-learn library for splitting the data into train-test samples, and for some basic model evaluation

For Model building and evaluation following libraries: sklearn.metrics import accuracy_score tensorflow.keras.optimizers import Adam sklearn.preprocessing import MinMaxScaler tensorflow.keras import Model, Sequential tensorflow.keras.layers import Dense, Dropout tensorflow.keras.losses import MeanSquaredLogarithmicError

Import following libraries from SKlearn : i) MinMaxscaler (sklearn.preprocessing) ii) Accuracy(sklearn.metrics) . iii) train_test_split (model_selection)

Import Following libraries from tensorflow.keras : models , layers,optimizers,datasets , and set to respective values. Grab to ECG.csv required dataset

Find shape of dataset

Use train_test_split from sklearn to build model (e.g. train_test_split(features, target, test_size=0.2, stratify=target)

Take usecase Novelty detection hence select training data set as Target class is 1 i.e.

Normal class

Scale the data using MinMaxScaler.

Create Autoencoder Subclass by extending model class from keras.

Select parameters as i)Encoder : 4 layers ii) Decoder : 4 layers iii) Activation Function : Relu iv) Model : sequential.

Configure model with following parametrs : epoch = 20 , batch size =512 and compile with Mean Squared Logarithmic loss and Adam optimizer.

e.g. model = AutoEncoder(output_units=x_train_scaled.shape[1]) # configurations of model model.compile(loss='msle', metrics=['mse'], optimizer='adam')

history = model.fit(x_train_scaled, x_train_scaled, epochs=20, batch_size=512, validation_data=(x_test_scaled, x_test_scaled)

Plot loss,Val_loss, Epochs and msle loss Find

threshold for anomaly and do predictions :

e.g. : find_threshold(model, x_train_scaled): reconstructions = model.predict(x_train_scaled) # provides losses of individual instances reconstruction_errors = tf.keras.losses.msle(reconstructions, x_train_scaled) # threshold for anomaly scores threshold = np.mean(reconstruction_errors.numpy()) \ + np.std(reconstruction_errors.numpy()) return threshold and then Get accuracy score

INPUT CODE:

```
import pandas as pd import numpy as np import
tensorflow as tf import matplotlib.pyplot as plt
import seaborn as sns from sklearn.model_selection
import train_test_split from sklearn.preprocessing
import StandardScaler
from sklearn.metrics import confusion_matrix, recall_score, accuracy_score, precision_score
RANDOM_SEED = 2021
TEST_PCT = 0.3
LABELS = ["Normal","Fraud"] dataset=pd.read_csv("creditcard.csv")
#check for any null values print("Any nulls in the
dataset",dataset.isnull().values.any()) print('-------')
print("No. of unique labels",len(dataset['Class'].unique()))
print("Label values",dataset.Class.unique())

#0 is for normal credit card transcation #1
is for fraudulent credit card transcation
print('-------')
print("Break down of Normal and Fraud Transcations")
print(pd.value_counts(dataset['Class'],sort=True))
#visualizing the imbalanced dataset
count_classes = pd.value_counts(dataset['Class'],sort=True)
count_classes.plot(kind='bar',rot=0)
plt.xticks(range(len(dataset['Class'].unique())),dataset.Class.unique())
plt.title("Frequency by observation number") plt.xlabel("Class")
plt.ylabel("Number of Observations")
#Save the normal and fradulent transcations in seperate dataframe
normal_dataset = dataset[dataset.Class == 0] fraud_dataset =
dataset[dataset.Class == 1]
#Visualize transcation amounts for normal and fraudulent transcations bins =
np.linspace(200,2500,100)
plt.hist(normal_dataset.Amount,bins=bins,alpha=1,density=True,label='Normal')
```

```
plt.hist(fraud_dataset.Amount,bins=bins,alpha=0.5,density=True,label='Fraud')
plt.legend(loc='upper right') plt.title("Transcation Amount vs Percentage of
Transcations") plt.xlabel("Transcation Amount (USD)") plt.ylabel("Percentage
of Transcations") plt.show() dataset sc = StandardScaler() dataset['Time'] =
sc.fit_transform(dataset['Time'].values.reshape(-1,1)) dataset['Amount'] =
sc.fit_transform(dataset['Amount'].values.reshape(-1,1)) raw_data =
dataset.values
#The last element contains if the transcation is normal which is represented by 0 and if fraud
then 1 labels = raw_data[:,-1]
#The other data points are the electrocadriogram data data = raw_data[:,0:-1]
train_data,test_data,train_labels,test_labels = train_test_split(data,labels,test_size =
0.2,random_state =2021) min_val =
tf.reduce_min(train_data) max_val =
tf.reduce_max(train_data)

train_data = (train_data - min_val) / (max_val - min_val)
test_data = (test_data - min_val) / (max_val - min_val)
train_data = tf.cast(train_data,tf.float32) test_data =
tf.cast(test_data,tf.float32) train_labels =
train_labels.astype(bool) test_labels =
test_labels.astype(bool) #Creating normal and fraud
datasets normal_train_data = train_data[~train_labels]
normal_test_data = test_data[~test_labels]
fraud_train_data = train_data[train_labels]
fraud_test_data = test_data[test_labels] print("No. of
records in Fraud Train Data=",len(fraud_train_data))
print("No. of records in Normal Train
Data=",len(normal_train_data)) print("No. of records in
Fraud Test Data=",len(fraud_test_data)) print("No. of
records in Normal Test Data=",len(normal_test_data))
nb_epoch = 50 batch_size = 64 input_dim =
normal_train_data.shape[1]
```

```
#num of columns,30 encoding_dim = 14 hidden_dim1
= int(encoding_dim / 2) hidden_dim2 = 4
learning_rate = 1e-7 #input layer input_layer =
tf.keras.layers.Input(shape=(input_dim,))
#Encoder encoder       =
tf.keras.layers.Dense(encoding_dim,activation="tanh",activity_regularizer            =
tf.keras.regularizers.l2(learning_rate))(input_layer) encoder =
tf.keras.layers.Dropout(0.2)(encoder) encoder =
tf.keras.layers.Dense(hidden_dim1,activation='relu')(encoder) encoder =
tf.keras.layers.Dense(hidden_dim2,activation=tf.nn.leaky_relu)(encoder)
#Decoder
decoder = tf.keras.layers.Dense(hidden_dim1,activation='relu')(encoder) decoder
= tf.keras.layers.Dropout(0.2)(decoder)
decoder = tf.keras.layers.Dense(encoding_dim,activation='relu')(decoder) decoder
= tf.keras.layers.Dense(input_dim,activation='tanh')(decoder
#Autoencoder autoencoder = tf.keras.Model(inputs =
input_layer,outputs = decoder) autoencoder.summary()
cp                                                                                                    =
tf.keras.callbacks.ModelCheckpoint(filepath="autoencoder_fraud.h5",mode='min',monitor='v
al_loss',verbose=2,save_best_only=True)
#Define our early stopping early_stop = tf.keras.callbacks.EarlyStopping(
monitor='val_loss',              min_delta=0.0001,              patience=10,
verbose=11,              mode='min',              restore_best_weights=True)
autoencoder.compile(metrics=['accuracy'],loss= 'mean_squared_error',optimizer='adam')
history = autoencoder.fit(normal_train_data,normal_train_data,epochs = nb_epoch,

batch_size = batch_size,shuffle = True,                    validation_data
= (test_data,test_data),
                    verbose=1,                    callbacks
= [cp,early_stop]).history
plt.plot(history['loss'],linewidth = 2,label = 'Train')
plt.plot(history['val_loss'],linewidth = 2,label = 'Test')
plt.legend(loc='upper right') plt.title('Model Loss')
plt.ylabel('Loss') plt.xlabel('Epoch')
```

```
#plt.ylim(ymin=0.70,ymax=1) plt.show()
test_x_predictions = autoencoder.predict(test_data)
mse = np.mean(np.power(test_data - test_x_predictions, 2),axis = 1) error_df
= pd.DataFrame({'Reconstruction_error':mse,
                'True_class':test_labels})
threshold_fixed = 50 groups =
error_df.groupby('True_class')

fig,ax = plt.subplots() for
name,group in groups:
    ax.plot(group.index,group.Reconstruction_error,marker='o',ms = 3.5,linestyle='',
label = "Fraud" if name==1 else "Normal")
ax.hlines(threshold_fixed,ax.get_xlim()[0],ax.get_xlim()[1],colors="r",zorder=100,label="
Th reshold") ax.legend() plt.title("Reconstructions error for normal and fraud data")
plt.ylabel("Reconstruction error") plt.xlabel("Data point index") plt.show()
threshold_fixed = 52 pred_y = [1 if e > threshold_fixed else 0 for e in
    error_df.Reconstruction_error.values] error_df['pred'] =
pred_y conf_matrix =
confusion_matrix(error_df.True_class,pred_y)
plt.figure(figsize = (4,4))
sns.heatmap(conf_matrix,xticklabels = LABELS,yticklabels = LABELS,annot
= True,fmt="d")

plt.title("Confusion matrix")
plt.ylabel("True class")
plt.xlabel("Predicted class") plt.show()
#Print Accuracy,Precision and Recall print("Accuracy
:",accuracy_score(error_df['True_class'],error_df['pred'])) print("Recall
:",recall_score(error_df['True_class'],error_df['pred'])) print("Precision
:",precision_score(error_df['True_class'],error_df['pred']))
```

OUTPUT

```
1  import pandas as pd
2  import numpy as np
3  import tensorflow as tf
4  import matplotlib.pyplot as plt
5  import seaborn as sns
6  from sklearn.model_selection import train_test_split
7
8  from sklearn.preprocessing import StandardScaler
9  from sklearn.metrics import confusion_matrix, recall_score, accuracy_score, precision_score
10
11 RANDOM_SEED = 2021
12 TEST_PCT = 0.3
13 LABELS = ["Normal","Fraud"]
```

```
C:\Users\DELL\anaconda3\lib\site-packages\scipy\__init__.py:146: UserWarning: A NumPy version >=1.1
ed for this version of SciPy (detected version 1.26.0
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
```

```
1  dataset=pd.read_csv("creditcard.csv")
```

Figure 1: Output 1

The above figure shows the importing of Libraries and uploading Dataset.

```
1  min_val = tf.reduce_min(train_data)
2  max_val = tf.reduce_max(train_data)
3
4  train_data = (train_data - min_val) / (max_val - min_val)
5  test_data = (test_data - min_val) / (max_val - min_val)
6
7  train_data = tf.cast(train_data,tf.float32)
8  test_data = tf.cast(test_data,tf.float32)
```

```
1  train_labels = train_labels.astype(bool)
2  test_labels = test_labels.astype(bool)
3
4  #Creating normal and fraud datasets
5  normal_train_data = train_data[~train_labels]
6  normal_test_data = test_data[~test_labels]
7
8  fraud_train_data = train_data[train_labels]
9  fraud_test_data = test_data[test_labels]
10 print("No. of records in Fraud Train Data=",len(fraud_train_data))
11 print("No. of records in Normal Train Data=",len(normal_train_data))
12 print("No. of records in Fraud Test Data=",len(fraud_test_data))
13 print("No. of records in Normal Test Data=",len(normal_test_data))
```

```
No. of records in Fraud Train Data= 389
```

Figure 2: Output 2

he above figure shows the training and testing of model.

```
autoencoder.compile(metrics=['accuracy'],loss= 'mean_squared_error',optimizer='adam')
```

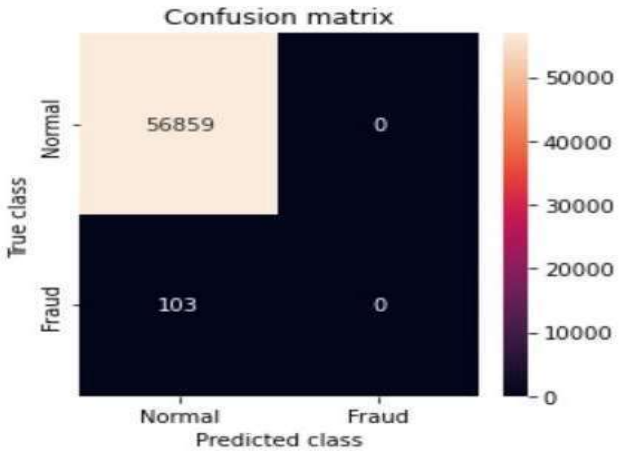
```
history = autoencoder.fit(normal_train_data,normal_train_data,epochs = nb_epoch,
                          batch_size = batch_size,shuffle = True,
                          validation_data = (test_data,test_data),
                          verbose=1,
                          callbacks = [cp,early_stop]).history
```

```
Epoch 1/50
3550/3554 [============================>.] - ETA: 0s - loss: 1.9525e-05 - accuracy: 0.0611
Epoch 1: val_loss did not improve from 0.00002
3554/3554 [==============================] - 20s 6ms/step - loss: 1.9526e-05 - accuracy: 0.0611 - val_loss: 2.0083e-05 - val
_accuracy: 0.0661
Epoch 2/50
3551/3554 [============================>.] - ETA: 0s - loss: 1.9523e-05 - accuracy: 0.0617
Epoch 2: val_loss did not improve from 0.00002
3554/3554 [==============================] - 18s 5ms/step - loss: 1.9523e-05 - accuracy: 0.0617 - val_loss: 2.0206e-05 - val
_accuracy: 0.0596
Epoch 3/50
3552/3554 [============================>.] - ETA: 0s - loss: 1.9514e-05 - accuracy: 0.0614
Epoch 3: val_loss did not improve from 0.00002
3554/3554 [==============================] - 18s 5ms/step - loss: 1.9511e-05 - accuracy: 0.0614 - val_loss: 2.0134e-05 - val
_accuracy: 6.1444e-04
Epoch 4/50
```

Figure 3: Output 3

The above figure shows Autoencoder of model using 50 epochs.

```
#Print Accuracy,Precision and Recall
print("Accuracy :",accuracy_score(error_df['True_class'],error_df['pred']))
print("Recall :",recall_score(error_df['True_class'],error_df['pred']))
print("Precision :",precision_score(error_df['True_class'],error_df['pred']))
```




```
Accuracy : 0.9981917769741231
Recall : 0.0
Precision : 0.0
```

Figure 4: Accuracy of model

The above figure shows Accuracy, Recall and Precision of model.

Conclusion:

Thus, the model is implemented and tested using 50 epochs for ECG Anomaly detection using Autoencoders.

The accuracy of the model after training is 99.8%